CHAPTER

6

THE SHELL AND TEXT FILES

After reading this chapter and completing the exercises, you will be able to:

- Describe how a Linux shell operates
- Customize a shell environment
- Use common text editors to create or modify text files
- Describe popular text-processing methods and tools used on Linux

In the previous chapter you learned about using a graphical system on Linux. You learned how to locate and configure the X Window System programs using a variety of utilities. You also learned how the X Window System is launched and how to modify the corresponding configuration files. You learned about the desktop environments that are commonly used on Linux and the graphical login application that can be used to start a graphical environment.

In this chapter you learn about how the Linux command-line environment—the shell—operates and how you can customize it to fit your preferences or those of other users on the Linux system. You learn about working with common Linux text editors to modify text files, and you are introduced to the more advanced programs used for complex text manipulation tasks.

UNDERSTANDING THE SHELL

In previous chapters you learned that the shell is the command interpreter, or command-line environment, for Linux. You learned a few basic commands such as `ls` to list files and `cp` to copy files. In the first part of this chapter you learn more about how the shell operates and how it interacts with the Linux kernel.

In many operating systems, a command interpreter is always running. A **command interpreter** is a program that accepts input from the keyboard and uses that input to launch commands or otherwise control the computer system. The most well known command interpreter is the `COMMAND.COM` program that is always running in DOS or Windows 95/98. This command interpreter is always available to receive input from the user via the keyboard or via programs that are running on the system. The command interpreter in DOS or Windows is integrated with the kernel of the operating system and provides functions that no other program can provide.

In Linux (or any UNIX-like system), the command interpreter (called the **shell**) has a very different relationship with the kernel and with users compared to the DOS/Windows model. The following list describes the major differences:

- The shell is only loaded when a user logs in or otherwise requests that a shell be launched.
- The shell is like any other program running on Linux. It has no special privileges, no special relationship with the Linux kernel, and no special capabilities.
- Different types of shells are available for Linux. A user can choose which shell best suits his or her preferences or environment.

As you learned in Chapter 4, when you first boot a Linux-based computer, the kernel starts the `init` program, which launches all of the system services that have been configured by the installation process or by the system administrator. The `init` program displays a login prompt (text based or graphical), but does not start a shell. Two reasons for the lack of a shell initially are:

- Linux is often used as a network server. Because no user is directly using the system (entering commands to launch programs), a shell is not needed in this situation. Instead, network services such as a Web server watch for incoming network requests and handle those requests appropriately.
- Linux security requires that no one can access the system until a valid username and password have been entered. Because a user cannot enter any commands until first logging in, no shell is required until a user has logged in.

Network services such as Web servers can be used to access files on Linux through a regular user account. This means a network service is governed by the permissions assigned to its user account. For example, suppose you are using a Web browser to request a particular file from the Web server. The Web server can only retrieve the requested file if its user account has permission to access that file. As a result, the Web server controls what the Web browser can access; you are not free to explore the files on the server as if you were logged in to the server.

After a user logs in from a character-mode screen, a default shell is started, which in turn provides a shell prompt where the user can enter commands. After the user logs in from a graphical login screen (using `xdm` or a similar program), the user does *not* see a shell prompt, because it is not strictly necessary. Instead, the user can manipulate the graphical environment (started by `xdm`) to access the same core functionality (launching programs and viewing files) that is provided by a shell. If you want to access the shell from a graphical environment, you need to use the appropriate menu command.

The Shell Prompt

Figure 6-1 shows a standard shell prompt that appears after logging in to Linux using a character-mode login prompt or after starting a shell from a graphical environment. The **shell prompt** is a set of words or characters indicating that the shell is ready to accept commands that you enter. The default shell prompt includes four components:

- The user account name that you used to log in (the first `nwells` in Figure 6-1).
- The hostname of the computer that you logged in to (`incline` in Figure 6-1).
- The last part of the full directory path for your current working directory (the second `nwells` in Figure 6-1, which is the last part of the full directory path `/home/nwells`).
- A prompt character (the ending `$` in Figure 6-1).

Figure 6-1 A shell prompt

Although you can alter the information provided in the shell prompt, the default setting shown here is usually an appropriate choice. The username and hostname help you keep track of your location within a networked environment involving many computers. The last part of your current working directory helps you keep track of your location within the directory structure. Although seeing the full path might be helpful in some cases, an extremely long directory path would be unwieldy; thus it makes sense to include only the last part of the path in the shell prompt.

The prompt character used in the standard Linux shell is a dollar sign, `$`. Other shells may use different prompt characters, such as a percent sign, `%`. On all shells, when you log in as `root` (the superuser), the prompt character changes to a hash mark, `#`. This makes it easier for you to determine as you work whether you have `root` permission or not. Remember that you should not use the `root` account unless you are completing system administration tasks.

The Functions of a Shell

The purpose of a shell is to make it easy for users to launch programs and work with files on the Linux system. That simple definition doesn't entirely capture the features of the shells you use in Linux, but it explains the basic rationale behind their design.

A shell's primary purpose is to launch programs. When you use the `ls` command to view the files in a directory, or use the `mv` command to rename a file, or use the `more` command to view a file, you are actually launching a program that performs those tasks. The shell processes the information entered at the keyboard and uses it to launch the program. In many cases, the information you enter on a command line includes parameters, such as the name of the file to copy and the location to copy it to. The shell passes these parameters to the program being launched. For example, entering the following command line at a shell prompt causes the shell to launch the `cp` command, handing it the two parameters `report.doc` and `report.doc.bak`. In this command, the `cp` command must decide what to do with the parameters, or return an error message if it cannot determine how to process the parameters.

```
cp report.doc report.doc.bak
```

If you enter the following command at a shell prompt, the shell will try to start a program called `report.doc` and hand that program the parameters `report.doc.bak` and `cp`. Because no program named `report.doc` exists, the shell will return an error message stating that it could not locate the requested command.

```
report.doc report.doc.bak cp
```

Besides the ability to start programs, the shell has many other built-in features that make it convenient to work with numerous files and commands on a Linux system. For example, from the shell, you can use keyboard shortcuts to enter long commands quickly, and you can control multiple programs that you have started from the shell prompt. In addition, you can define variables (assign numbers or strings to a name) to make your shell environment easier to use or to provide information (the values of variables) that other programs besides the shell can access when needed. Many of these features are described in this chapter; others are described later in the book.

A particularly important feature of a Linux shell is that it gives users the ability to write scripts (or programs) that the shell can execute. As you will learn in Chapter 12, a script is essentially a list of commands stored in the form of a text file. Instead of entering each of these commands, one by one, at the command line, you can use a script to automate the execution of a series of commands. Chapter 12 is dedicated to teaching you how to write shell scripts.

Different Types of Shells

When UNIX was first created decades ago, the original developers decided that the shell (the command interpreter) should be separated from the operating system so that it could be changed or improved later without affecting the operating system. As described in the previous section, the shell is just a regular program whose purpose is to launch other programs. The original shell for UNIX, written by Stephen Bourne, is called the **Bourne shell**. The Bourne shell program is called `sh` (for *shell*). Although the Bourne shell is standard on

all UNIX and Linux systems, it is an old program with limited functionality (it was first written nearly 30 years ago).

True to the foresight of the developers of UNIX, other developers started with the Bourne shell and altered or enhanced it to provide new functionality. These later-generation shells are used on all UNIX and Linux systems today. Table 6-1 shows the commonly available shells for Linux.

Table 6-1 Linux Shells

Shell name	Program name	Description
Bourne shell	`sh`	The original UNIX shell. The `sh` program on Linux usually refers to the `bash` program. `bash` contains all `sh` functionality, plus interactive features such as history and tab completion (described later in this chapter) and shell programming via shell script files.
C shell	`csh`	A shell developed by Bill Joy in the 1970s. He focused on adding easy-to-use features for interactive work at the shell prompt. The C shell was the first to contain features similar to history and tab completion; these features were later added to the `bash` shell and other shells as well. The C shell uses a more complex syntax for shell programming than the Bourne and `bash` shells. Because of this, it is not popular for shell programming, though its interactive features make it popular with users who are not creating shell programs.
TENEX/TOPS C shell (also called the **TC shell**)	`tcsh`	An enhancement of the C shell. This is the version of the C shell that is commonly used on Linux systems.
Korn shell	`ksh`	A proprietary (not freely available) shell written by David Korn. The Korn shell is a revision of the Bourne shell that includes the interactive features of the C shell but maintains the Bourne shell programming syntax, which is considered easier to use than C shell programming syntax.
Public Domain Korn shell	`pdksh`	A version of the Korn shell that is freely available. (This shell is often accessed using the program named `ksh` on Linux systems.)
Bourne Again shell	`bash`	An enhanced and extended version of the Bourne shell created by the GNU project for use on many UNIX-like operating systems. Commonly referred to as the `bash` shell, rather than by its full name, `bash` is the default Linux shell.
Z shell	`zsh`	A recently developed shell that combines Korn shell interactive features with the C shell programming syle (for those who prefer the more complex syntax of the C shell).

The default shell for all Linux systems is `bash` (pronounced as the word looks, "bash"). Users on a Linux system are normally content to use the `bash` shell exclusively. The exception occurs when a user has experience with another type of shell from working on other UNIX systems, or the user writes a lot of shell scripts and needs the features of another shell. (The C shell and TC shell both use different shell programming methods than `bash`.)

Shells can be roughly divided into two groups based on the type of shell programming commands used. The two groups are:

- Those that follow the Bourne shell programming style (which is based on a very old programming language called ALGOL)
- Those that follow the C shell programming style (which is based on the widely used C language)

Further shell derivatives have combined features from different shells to make this grouping less distinct. For example, the Z shell includes many popular features of the `bash` shell but with C shell–style programming. But the overall distinction between these two groups is still valid.

Not all of the shells in Table 6-1 will be installed by default or even included on the CD for all Linux distributions. Contact your Linux vendor or an Internet download site such as *www.linuxberg.com* to obtain a particular shell that is not included on your Linux CD.

In Linux, the shell started for each user is determined by the settings in the user account configuration file. Chapter 8 describes how you can set up or modify this configuration file. If the shell you want to use is installed on the Linux system, changing to a new default shell is very easy using the `usermod` command described in Chapter 8. Each user on the system can select a preferred shell independent of all other users.

To immediately run a different shell that is installed on the Linux system you are using, enter the name of that shell program. For example, if you are working in the standard `bash` shell but you want to run the C shell instead, enter this command and you are immediately switched to the C shell:

```
csh
```

Entering Commands

Modern shells like the `bash` and the Korn shell include features designed to simplify the process of entering commands and command parameters. Two of the most useful features are tab completion and history, which are described in the following sections.

Using Tab Completion

Tab completion is a shell feature that lets you enter part of a file or directory name and have the shell fill in the remainder of the name. Using tab completion makes it easier to enter long or complex directory paths and filenames. This is often helpful because Linux filenames

can be very long, and they sometimes include punctuation, multiple digits or periods, and mixed upper- and lowercase. Because tab completion is a feature of the shell, it works whenever you are entering text at a shell prompt, no matter which command you are entering. Anytime the shell determines that you are trying to enter a command name, a filename, or a directory name, you can use tab completion.

To see how tab completion works, consider this example. Suppose you want to use the `rpm` command to install a new software package that you have downloaded and placed in the `/tmp` directory. The filename of the package is shown here:

```
desktop-backgrounds-1.1.2-6.noarch.rpm
```


To install this package, you enter the `rpm` command followed by the path and filename of the package. But for this example, suppose you just enter the following:

```
rpm -Uvh /tmp/deskt
```

To take advantage of tab completion at this point, you press the Tab key. The shell then looks at the contents of the `/tmp` directory for a file or subdirectory matching the first few letters you typed (`deskt`). Once it finds the package name, the shell fills in the remaining filename; thus, immediately after pressing Tab, you see this at the command line:

```
rpm -Uvh /tmp/desktop-backgrounds-1.1.2-6.noarch.rpm
```

Now suppose the `/tmp` directory contains another file named `desktop`. Instead of filling in the full filename when you press Tab, the shell beeps to indicate that a unique matching name is not available. You can then press Tab a second time to have the shell display all of the matching names, like this:

```
rpm -Uvh /tmp/deskt
desktop   desktop-backgrounds-1.1.2-6.noarch.rpm
```

After reviewing this list of available files with similar names, you can type enough of the name to make it unique, and then press Tab again to fill in the complete filename. In this case, because the hyphen is the first character that distinguishes the two filenames, you would need to enter the following:

```
rpm -Uvh /tmp/desktop-
```

When you first use tab completion, you may think it's more work than it's worth to keep pressing Tab and entering a few more letters if the filename is not unique. But after some practice, using tab completion to enter long filenames or paths becomes almost automatic—much easier than entering the complete file or directory name manually.

When the first part of the name that you enter is a directory, tab completion fills in the directory name, ending with a forward slash. This means you can immediately begin typing the name of a subdirectory or file within that directory. You will have a chance to try using tab completion in Project 6-1, at the end of this chapter.

Using the History Feature

A second shell feature designed to make launching commands easier is the history feature. The **history feature** records a list of each command that you enter at the shell prompt. You can quickly call up and repeat any command from this list without entering the command again.

The simplest method of accessing the history list of commands is to press the Up arrow key. This displays the most recently executed command on the command line. To use the command, press Enter. Pressing the Up arrow key repeatedly displays in turn each of the previously entered commands (the commands in the history list). Hold down the Up arrow key to see dozens of commands flash by at the shell prompt (the full contents of the history list). Press the Down arrow key to execute commands farther down in the history list (those more recently entered).

When the command you want to repeat was entered some time ago, using the Up arrow key to locate it in a large history list can be tedious. In this situation, the `history` command is useful. The **`history`** command displays the entire **history list**, which contains the most recently executed commands. (Normally at least 100 commands are included in the history list.) The following shows the last few lines of a history list. (Of course, the commands and numbers in the history list on your system will differ from this sample output.)

```
33  who
34  vi /etc/passwd
35  gimp
36  cd /etc
37  cd X11/
38  cd xdm
39  more Xsession
40  rpm -qa |grep XFree
41  mount -t ext2 /dev/hda3 /mnt/openlinux/
42  mcopy /mnt/openlinux/etc/XF86Config A:
43  file Xwrapper
44  umount /mnt/cdrom
45  type fvwm
46  exit
47  clear
48  mv ch05/ch04fig.zip ch04/
49  mv ch04/ch04fig.zip course_ch04/
50  cd course_ch04/
```

The length of this list can make it difficult to quickly locate the command you want to reuse. Thus you may prefer to use one of the following three methods for locating previously executed commands:

- Use the history number
- Use the beginning of a command
- Search the history list

The term *history number* refers to the number to the left of each item in the history list. For example, in the history list above, the first item has a history number 33.

You can execute any of the commands in the history list by entering the number of that command prefixed by an exclamation point. An exclamation point is sometimes called a **bang** in UNIX and Linux. So to execute the most recent `mount` command, you would enter `!41` (pronounced "bang-forty-one").

The shell displays the command matching that number and immediately executes it. You don't need to view the history list before using this technique if you already know the number of the command you want to execute. But be aware that the numbers change as you enter new commands.

To use the command name to repeat a command, use an exclamation point followed by the first part of the command you want to repeat. In the sample `history` output shown, you could execute the most recent `mount` command using this command:


```
!mou
```

When you execute this command, the shell searches for the most recent command that begins with the letters "mou" and executes it.

When executing a command from the history list using any of the three methods described, remember that the commands are executed from your current working directory, which may be different from the directory where they were originally executed. If a command does not include a full pathname, you might see unexpected results. Be especially careful when using the partial command name method to reexecute a command without checking the full text of the command.

You can also search the history list without reexecuting a command to see what the command parameters were or how you completed a task. This method requires the use of a pipe symbol and the `grep` command, both of which are discussed in detail in Chapter 7. To use this method, enter the `history` command followed by the `grep` command and the command name you want to locate. For example, in the sample `history` output shown previously, suppose you want to search for a `mount` command to see what parameters it contains. The following command will display all items in the history list that contain the `mount` command. You can then review the displayed output to learn about the previously executed command.

```
history | grep mount
```

The `bash` shell supports additional techniques for executing commands. But the examples shown here for using tab completion and the `history` command should help you enter commands much more efficiently in Linux. To learn more about these features, review the online manual page for the `bash` shell by entering `man bash`.

The Shell Start-up Process

You learned in Chapter 5 that certain scripts are executed each time the graphical system of Linux is launched. In a similar manner, several scripts are executed when you log in to Linux or start a new shell. These scripts initialize (or configure) various parts of a user's environment, as described in the next section of this chapter.

When a user first logs in to a Linux system, the script `/etc/profile` is executed. The **`/etc/profile`** script contains configuration information that applies to every user on the Linux system. Each user's home directory can contain another start-up script called **`.profile`** (with an initial period). The `.profile` script in a user's home directory is also executed when the user logs in, but the `.profile` script is specific to a single user. Each user's home directory can contain a different `.profile` script. Only the root user can change the `/etc/profile` script; any user can change the `.profile` script in his or her home directory.

On some systems, additional scripts are executed when a user logs in. For example, on Red Hat Linux, a set of scripts located in the `/etc/profile.d` directory is started by the `/etc/profile` script. The scripts in `/etc/profile.d` add specific configuration information for KDE, a language selection, or other system features set up by the installation program. Red Hat Linux also uses a file called `.bash_profile` rather than the standard `.profile` script in each user's home directory. The operation of the file is the same as the `.profile` script.

The `profile` scripts are executed when a user logs in; additional scripts are executed when a user starts a shell. Because a shell is started immediately when a user logs in at a character-mode screen, these additional scripts are generally executed immediately after the profile scripts. A user working in a graphical environment can start multiple `bash` shells without logging in to Linux again. When a new shell is started by a user who is already logged in, the additional scripts described next (such as `/etc/bashrc`) are executed; however, the `profile` scripts are not executed again.

Some Linux distributions, including Red Hat Linux, provide an **`/etc/bashrc`** script that is executed for all users on the system each time a `bash` shell is started. Other Linux distributions rely on the `/etc/profile` script for configuration settings that should apply to all users, though this script is only executed at the time a user logs in.

Each user's home directory contains a script called `.bashrc`. The **`.bashrc`** script is executed each time the user starts a `bash` shell. Any configuration information that a user wants to add to his or her environment can be placed in the `.bashrc` file. Additional scripts with similar names are sometimes used on a Linux distribution. Examples include the following:

- `.bash_default`, which is executed each time a `bash` shell is started
- `.bash_login`, which is executed each time a `bash` shell is started
- `.bash_logout`, which is executed each time a user closes a `bash` shell

You may find scripts on your Linux distribution. In general, the names of these files provide a good description of when they are used. Consult your Linux vendor or try placing test commands in each file (see Chapter 12 for directions) if you are unsure of how the files are used. Figure 6-2 shows how a typical start-up script works when a user logs in to Linux.

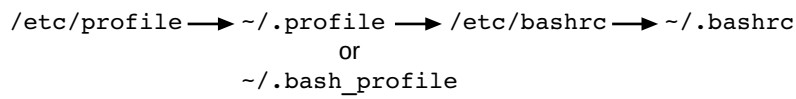

Figure 6-2 A typical series of start-up scripts when logging in to Linux

When the user logs in or starts a shell, the systemwide script file is executed, followed by the corresponding file in the user's home directory. When starting the X Window System (see Chapter 5), the system default file `/etc/X11/xinit/xinitrc` is executed *only* if the `xinit` program does not find an `.xinitrc` file in the user's home directory.

The scripts described here apply when a `bash` shell is started. Similar files are executed when a user chooses to work with a C shell, Korn shell, or other shell. For example, a user's home directory may contain a file called `.cshrc` or `.kshrc`. These scripts would be executed each time a C shell or Korn shell was started, respectively. Because the script format is different for each type of shell, different script files are needed to initialize each shell. These configuration scripts can coexist in a user's home directory (and in the `/etc` directory, for systemwide configuration files), each one being executed only when the corresponding shell is launched.

CUSTOMIZING THE SHELL

The configuration scripts that Linux executes when a user logs in, or that `bash` executes when a shell is launched, provide a place where users can customize the Linux environment in which they work. The following sections describe several methods of customizing the shell environment. These features are separate from any customization that a user or system administrator may choose to do within a graphical environment.

Using Aliases

An **alias** is a string of characters that is substituted for another string of characters at the shell prompt. The general format of the `alias` command looks like this:

```
alias <string entered by user>=<string substituted by the shell>
```

For example, suppose that you are accustomed to entering the `COPY` command in DOS to copy files. Rather than enter the correct Linux command (`cp`) each time you want to copy files in Linux, you can create an alias that allows you to use the DOS command. You create the alias as follows:

```
alias copy=cp
```

With this alias in effect, each time you enter the string `copy` at the shell prompt, the shell replaces it with the string `cp`. In effect, you can now enter a command like this:

```
copy /tmp/download.tgz /home/nwells/
```

Because of the alias created earlier, the preceding command would then execute the following command:

```
cp /tmp/download.tgz /home/nwells/
```

After you create an alias, each time you enter text at a command prompt, the shell substitutes one string of characters for the other that you defined. You must be careful when you create an alias that uses an existing command name. For example, entering the command `alias more=less` would render the `more` command inoperative, because every time you entered `more`, the shell would substitute the string `less`.

To see a list of aliases that are in effect as you work, enter the `alias` command without any text after it. Many Linux distributions include a few aliases such as the following ones. These are defined for all users in a start-up script, such as `/etc/profile` or `/etc/bashrc`. Notice that when the string substituted by the shell contains a space, it must be enclosed in quotation marks.

```
alias ll="ls -la"
alias rm="rm -i"
alias cp="cp -i"
```

The `alias` command is useful in several circumstances, including those listed here:

- Aliases can shorten long commands. For example, if you regularly enter a command with many options, create an alias so you can enter that command with just two or three characters.
- Aliases can correct typing or spelling mistakes. For example, if you always enter `sl` instead of `ls`, you can create an alias that makes `sl=ls`. Aliases can help people new to Linux use the system without knowing all of the commands perfectly.
- Aliases can protect you from erasing files by automatically inserting options with commands that are used to delete files. For example, the `alias` command shown previously for the `cp` command (`alias cp="cp -i"`) causes the shell always to execute the `cp` command with the `-i` option, which prevents overwriting files when copying files.
- Aliases can add command names that you prefer to use, but that are not part of Linux by default. For example, you can use an alias to substitute the string `mv` for `ren`. (The DOS command `ren` does not exist in Linux; the `mv` command is used instead.)

Of course, when you use aliases for these purposes, you won't master Linux commands, nor will you improve your typing skills. But used wisely, aliases can make tasks proceed more quickly as you work at a Linux command line. Entering the `alias` command causes that alias to be active only as part of the current shell. Once you have decided on several `alias` commands that suit your needs, add those commands to the `.bashrc` file in your home directory so that they are executed each time you start a shell.

Symbolic Links

Symbolic links are a feature of the Linux file system. They are not part of the shell, but they can make working in the shell easier. Symbolic links are also sometimes confused with aliases in the shell, so they are presented here to avoid that confusion. Chapter 9 describes symbolic links in more technical detail.

A **symbolic link** is a file that refers to another filename rather than to data in a file. For example, suppose several employees in a company want to work on the same file. The system administrator can place the file in a directory and then create a symbolic link in each user's home directory to access the real file. If the real file is `/tmp/report.doc`, the symbolic links might be `/home/nwells/report.doc`, `/home/davis/newreport.doc`, and `/home/laura/report.doc`.

All three users can access the same physical file, `/tmp/report.doc`, by opening the respective files in each home directory. The file system follows the symbolic link to the file that it points to and opens that file. When users make changes after opening the file in their home directory, they are all changing the same file. Figure 6-3 illustrates a symbolic link.

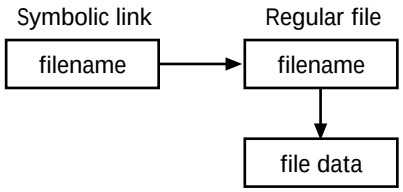

Figure 6-3 Symbolic link referring to another file

Symbolic links are used when the same data must be accessed from two locations in the directory structure, or by two (or more) different names. Using a symbolic link takes only a few bytes of hard disk space—enough to store the filename that the link refers to rather than a copy of the file. Symbolic links are commonly used in directories such as `/lib` and `/usr/lib`, where a system file must be referred to by several names in order for programs to find it.

You can view any symbolic links in a directory by using the `ls -l` command. For example, part of the output from the command `ls -l /lib` is shown below. Within the first column, a symbolic link is indicated by the letter *l*. In the last column, the filename preceded by an arrow (–>) is the file to which the symbolic link points.

```
-rwxr-xr-x   1 root     root      4016683 Apr 16  1999 libc-2.1.1.so
lrwxrwxrwx   1 root     root           13 Nov 18 02:35 libc.so.6 -> libc-2.1.1.so
lrwxrwxrwx   1 root     root           17 Nov 18 02:36 libcom_err.so.2 -> libcom_err.so.2.0
-rwxr-xr-x   1 root     root         7889 Mar 21  1999 libcom_err.so.2.0
-rwxr-xr-x   1 root     root        63878 Apr 16  1999 libcrypt-2.1.1.so
lrwxrwxrwx   1 root     root           17 Nov 18 02:35 libcrypt.so.1 -> libcrypt-2.1.1.so
-rwxr-xr-x   1 root     root       787688 Apr 16  1999 libdb-2.1.1.so
lrwxrwxrwx   1 root     root           15 Nov 18 02:35 libdb.so.2 -> libdb1-2.1.1.so
lrwxrwxrwx   1 root     root           14 Nov 18 02:35 libdb.so.3 -> libdb-2.1.1.so
-rwxr-xr-x   1 root     root       219002 Apr 16  1999 libdb1-2.1.1.so
lrwxrwxrwx   1 root     root           15 Nov 18 02:35 libdb1.so.2 -> libdb1-2.1.1.so
-rwxr-xr-x   1 root     root        73486 Apr 16  1999 libdl-2.1.1.so
lrwxrwxrwx   1 root     root           14 Nov 18 02:41 libdl.so.1 -> libdl.so.1.9.5
-rwxr-xr-x   1 root     root         5388 Mar 21  1999 libdl.so.1.9.5
lrwxrwxrwx   1 root     root           14 Nov 18 02:35 libdl.so.2 -> libdl-2.1.1.so
lrwxrwxrwx   1 root     root           13 Nov 18 02:36 libe2p.so.2 -> libe2p.so.2.3
-rwxr-xr-x   1 root     root        14519 Mar 21  1999 libe2p.so.2.3
lrwxrwxrwx   1 root     root           16 Nov 18 02:36 libext2fs.so.2 -> libext2fs.so.2.4
-rwxr-xr-x   1 root     root        84999 Mar 21  1999 libext2fs.so.2.4
-rwxr-xr-x   1 root     root       538944 Apr 16  1999 libm-2.1.1.so
lrwxrwxrwx   1 root     root           13 Nov 18 02:35 libm.so.6 -> libm-2.1.1.so
lrwxrwxrwx   1 root     root           15 Nov 18 02:42 libncp.so -> libncp.so.2.2.0
lrwxrwxrwx   1 root     root           15 Nov 18 02:42 libncp.so.2 -> libncp.so.2.2.0
-rwxr-xr-x   1 root     root        96737 Apr  6  1999 libncp.so.2.2.0
```

Graphical file manager windows (such as those in Gnome and KDE) also indicate a symbolic link using a special icon. Figure 6-4 shows a Gnome file manager window containing the `/lib` directory. The small arrow in the lower-right corner of the file's icon marks a file as a symbolic link.

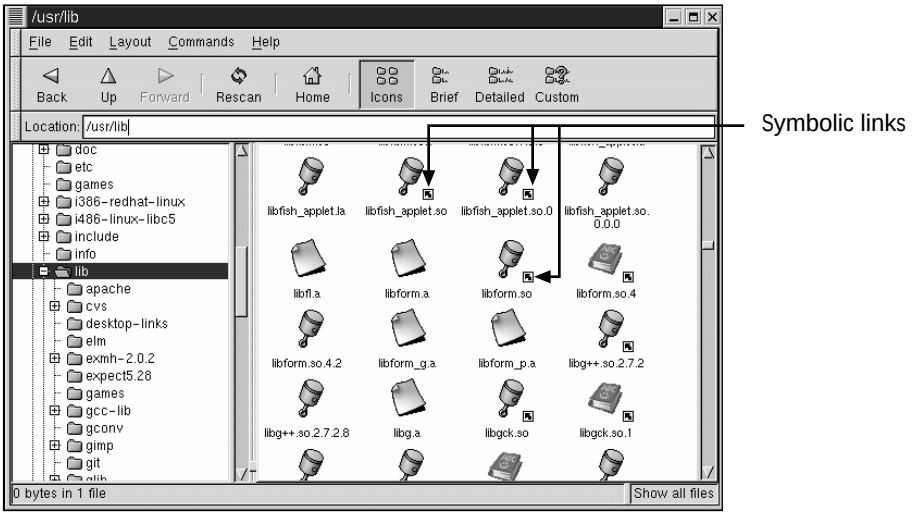

Figure 6-4 Symbolic links in a Gnome file manager window

You use the **`ln`** command with the **`-s`** option to create a symbolic link. The syntax of this command is as follows:

```
ln -s <existing file> <symbolic link to create>
```

For example, if you have a file called `report.doc` in your home directory and you want to create a symbolic link to the `/tmp` directory named `newreport.doc`, you would use the following command:

```
ln -s /home/nwells/report.doc /tmp/newreport.doc
```

In the command above, you could use relative pathnames depending on your current working directory.

Don't confuse aliases and symbolic links. An alias causes the shell to substitute a different string in text that you enter. A symbolic link causes the file system to pass a request for one file to a different file in the directory structure.

Environment Variables

Environment variables are settings, or values, available to any program launched by a particular user. Each user has a separate set of environment variables available to programs launched by that user. In Linux terminology, an environment variable is assigned a value. For example, the value of the `HOME` environment variable is the path to a user's home directory.

The USER environment variable is assigned a value of the current user account. The OSTYPE environment variable is assigned a value of the operating system type, Linux.

Environment variables define the environment in which a user works. For example, the HOME variable contains the value of the user's home directory path, such as /home/nwells. When the cd command is executed without a parameter, the shell changes the current working directory to the value of HOME.

The initialization scripts or start-up scripts (that are run when Linux is booted or when a user logs in) create many environment variables and assign values to them. Each time a user starts a program, the environment of that new program is taken from (inherited from) the program that launched it, which is normally the shell that the user is working in. The shell has many environment variables defined. These variables are created by various scripts that are executed during system boot and when a user logs in. When you start a new program, all of those environment variables become part of the environment in which the new program runs. This means that the new program you launch can access the values of all those environment variables. For example, any program can request the value of the USER environment variable to see which user launched the program.

The **set** command displays a list of all environment variables defined in your current environment. The output of the set command on a Red Hat Linux system is shown below. Many variables listed by set are used by system processes with which you are not yet familiar, but you will recognize some of them. For example, the PWD=/home/nwells line indicates the current working directory. When you execute the pwd command, the value of this environment variable is printed to the screen. When you use the cd command, the value of this variable is updated to a new directory name.

```
BASH=/bin/bash
BASH_ENV=/root/.bashrc
BASH_VERSION=1.14.7(1)
COLUMNS=80
EUID=0
HISTFILE=/root/.bash_history
HISTFILESIZE=1000
HISTSIZE=1000
HOME=/root
HOSTNAME=incline
HOSTTYPE=i386
IFS=
INPUTRC=/etc/inputrc
KDEDIR=/usr
LINES=25
LOGNAME=root
MAIL=/var/spool/mail/root
MAILCHECK=60
OPTERR=1
OPTIND=1
OSTYPE=Linux
PATH=/usr/bin:/sbin:/bin:/usr/sbin:/usr/bin:/usr/X11R6/bin:/root/bin
```

```
PPID=604
PS1=[\u@\h \W]\$
PS2=>
PS4=+
PWD=/root
SHELL=/bin/bash
SHLVL=1
TERM=linux
UID=0
USER=root
USERNAME=
_=set
```

You can also view the value of a single environment variable using the **echo** command, which prints text to the screen. You can include an environment variable name after the `echo` command to print the value of that variable. The variable name is preceded by a dollar sign so that the value of the variable is substituted by the shell. For example, to print the value of the `HOME` variable to the screen, use the following command:

```
echo $HOME
```

Many programs use environment variables to obtain information about your environment or about how the program should function. For example, a program may use the `HOME` variable to determine where to look for a user's data files. A program may also expect that certain environment variables have been set up specifically for the use of that program. For example, the documentation for a database program may state that before launching the program, you must define an environment variable named `DB_DIR` that defines the directory where the database files are located. If you execute the database program without first setting this environment variable, the program will not function correctly. (In such a case, the program usually displays an error message indicating that you must set a certain environment variable.) When programs need certain environment variables set, you should include a command to set those variables either in the systemwide start-up scripts or in a specific user's start-up scripts (if only one user runs the program in question).

You use the **export** command to make a newly created environment variable available to other programs running in the same environment. For example, you can define a new environment variable for the example database program just mentioned, and then make that variable available to the database program, using these two commands:

```
DB_DIR=/usr/local/db_data
export DB_DIR
```

Most environment variables are all uppercase letters, but they are case sensitive. If a program requires that you set up an environment variable, follow the format given in the program's documentation.

An excellent example of a program that uses environment variables is the shell itself. The online manual page for the `bash` shell lists dozens of variables that the shell uses (or can use,

if you set them) to control or select features of the shell. Two of these variables deserve mention here as examples.

The **PATH** environment variable contains a list of directories on the Linux system that the shell searches each time a command is executed. When you enter a program name to launch at the shell prompt, the shell searches in each directory listed in the value of the PATH variable. If the program is not found in the first directory, the second is searched, and so forth. The command to view the value of PATH is `echo $PATH`. Sample output of this command on a Red Hat Linux system is shown below. (The value of PATH varies depending on whether you are logged in as root or as a regular user. The output here is for a regular user account.)

```
/usr/bin:/bin:/usr/bin:/usr/local/bin:/usr/bin/X11:/usr/X11R6/
  bin:
/home/nwells/bin
```


When you want to execute any program or script that is not located in a directory that is part of the PATH variable, you must provide the shell with the file's complete pathname. For example, you must give the full pathname to execute any program in your home directory. If the program is named `newprogram` and your current working directory is `/home/nwells`, either of these commands will run the program:

```
/home/nwells/newprogram
./newprogram
```

If you simply enter `newprogram` alone, the shell will look in the PATH directories and be unable to find the `newprogram` program. An interesting exercise is to press the Tab key twice on an empty shell prompt line. The shell then attempts to use tab completion, but because you have entered no characters, the list of possible matches is very large, and the shell requests confirmation with a message like this one:

```
There are 1599 possibilities. Do you really
wish to see them all? (y or n)
```

Pressing the Y key for yes causes the shell to list all of the executable programs that it can find in the PATH directories.

Another example variable used by the shell is called `PS1`. This variable defines the shell prompt for `bash`. The command `echo $PS1` produces the following output:

```
[\u@\h \W]\$
```

The `\u`, `\h`, and `\W` parameters refer to the username, hostname, and working directory, respectively. You can alter the shell prompt by changing the value of this variable. You will have a chance to try using this command in Project 6-1, at the end of this chapter.

USING TEXT EDITORS

The most often-used tool of a Linux system administrator is a text editor. As you have probably already noticed from the discussion so far, most of what happens on a Linux system is controlled by a text configuration file. Graphical configuration utilities are sometimes available to assist with

configuration, but a competent Linux system administrator can also modify the configuration files using any text editor. This provides the flexibility to update or repair a Linux system without having access to special configuration utilities.

A Variety of Editors

Linux supports numerous text editors; at least three are included with every popular version of Linux. Some of these text editors are graphical, such as the KDE text editor shown in Figure 6-5. You don't need any special training to use graphical text editors because a menu bar and dialog boxes guide you through any editing tasks you need to perform. The disadvantage of a graphical editor is that it requires you to be in the X Window System, which is not always available. (You may also choose not to use X because it consumes a lot of system resources.) At those times when a graphical environment is not available, you must use a text-based editor.

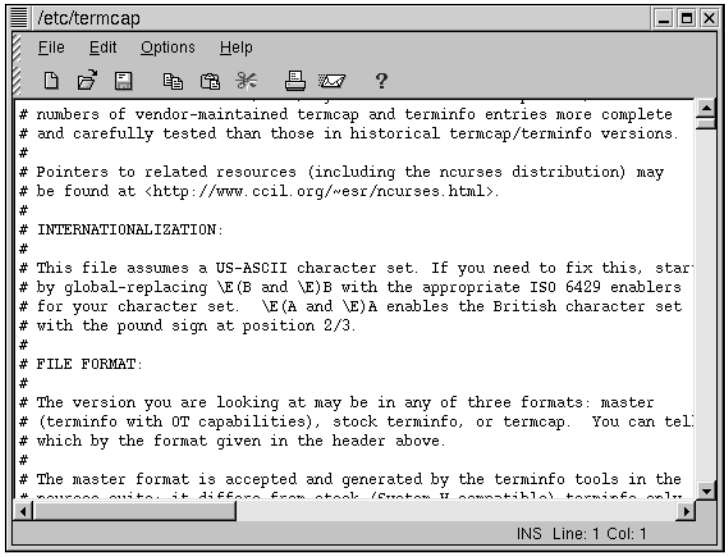

Figure 6-5 The KDE text editor

The following list shows some of the better-known character-mode text editors included with various Linux distributions. Not all are included with every version of Linux, but several are probably available on the Linux system you are using.

- `vi`: the name stands for *visual editor*, though you may wonder if this title is appropriate the first time you interact with `vi` because it doesn't provide any visual clues about how to use the editor's functions. This is the most widely used editor on UNIX and Linux systems. It is discussed in detail in the next section. Different versions of `vi`, such as `vim` and `elvis`, are usually launched with the command `vi`.

- `emacs`: this powerful editor provides macros, programming tools, customization, and hundreds of keyboard shortcuts. A graphical version called `xemacs` is available.
- `pico`: this simple editor includes on-screen information about which Control key sequences perform which functions.
- `ed`: because this is a line editor, you can only work with one line of text at a time, instead of viewing an entire screen full of information at once as you can in other text editors.
- `joe`: this is another simple text editor with on-screen command help.

Figure 6-6 shows the `pico` editor with a text file loaded for editing. Notice the commands at the bottom of the screen. Each item indicates a control character that you can use to control the editor. For example, the text `^X Exit` indicates that you can press Ctrl+X to exit `pico`.

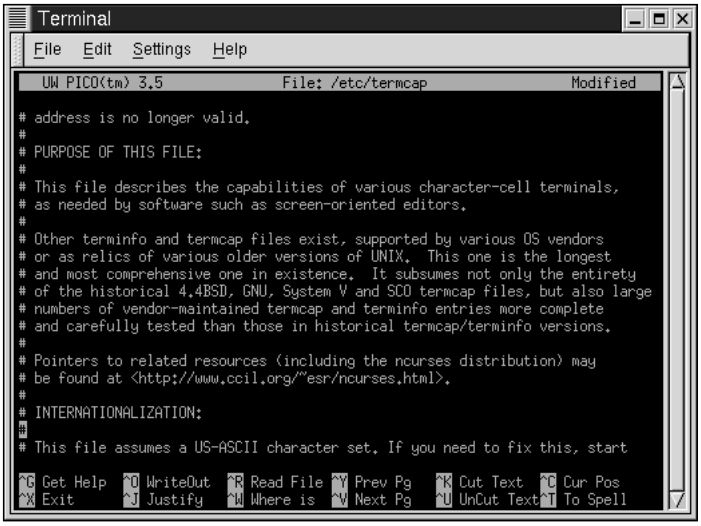

Figure 6-6 The `pico` text editor

Using the `vi` Editor

Because the `vi` editor is a powerful program that is available on all Linux systems, it is important that all system administrators have at least a basic familiarity with it. `vi` is not easy to learn, however, because it requires you to memorize strange key sequences to perform even basic commands. Once you have learned a few commands, the patterns used by `vi` start to emerge, and learning new commands becomes easier.

To launch the `vi` editor, enter the command `vi` at any Linux shell prompt. You can include the name of a file you want to edit after the program name, such as `vi /etc/lilo.conf`, or just use the program name to begin creating a new file. When you open a new file, you see tilde characters (~) down the left side of the screen. These indicate lines that are not part

of the file (because a new file is empty). Figure 6-7 shows `vi` after starting it without specifying a file to edit.

Figure 6-7 A new file in the `vi` editor

`vi` is a modal editor. In a **modal editor**, your keystrokes are interpreted differently depending on the task at hand. Different modes (such as command mode and edit mode) determine how keystrokes are interpreted by the editor. For example, if you are in command mode and press a key, the key is interpreted as a command; if you are in edit mode and press a key, the key is interpreted as data entry and is added to the document. `vi` has several modes. The most important ones are listed here:

- Command mode: keystrokes are interpreted as commands to edit the file, such as deleting lines or searching for text.
- Insert mode: keystrokes are inserted into the document you are creating.
- Replace mode: keystrokes are added into the document you are creating, overwriting any existing text at the place where you begin typing.

When you open `vi`, you begin in command mode. You can always return to command mode by pressing the Esc key. When you are in command mode, `vi` displays only the document you are editing. When you are in insert mode or replace mode, you see a message line at the bottom of the screen with the text `--INSERT--` or `--REPLACE--`. (You'll learn how to switch to another mode later in this section.)

Many commands in `vi` require you to enter a series of keystrokes. The following tables use the notation "Ctrl+X" to indicate "hold down the Ctrl key while pressing the X key." The notation "1, Ctrl+g" indicates "press the 1 key, then hold down the Ctrl key while pressing the g key." All `vi` commands are case sensitive. The notation "Ctrl+g" indicates a lowercase *g*. The notation "Ctrl+Shift+G" indicates an uppercase *g*.

You can use the arrow keys and the Page Up and Page Down keys to move around the screen as you edit a document. These keys normally work if you are in insert mode or replace mode as well. Table 6-2 shows additional commands you can use to move around a large document while you are in command mode.

Table 6-2 `vi` Commands Used for Moving Around a Document

Keystroke	Description
j	Move the cursor one line down.
k	Move the cursor one line up.
h	Move the cursor one character left.
l	Move the cursor one character right.
w	Move the cursor one word forward.
b	Move the cursor one word backward.
Shift+G	Move to the last line of the file.
1, Shift+G	Move to the first line of the file.
Ctrl+g	Display a status line at the bottom of the screen to indicate the line number where the cursor is positioned and the name of the file being edited.

If you are working on Linux over a network connection (for example, with the Microsoft Windows telnet program), `vi` may have trouble displaying text correctly. The first indication of a problem is usually that the arrow keys do not work correctly. You can still use the commands in Table 6-2 to move around the document, but you may want to investigate getting a different terminal program for the Windows system, such as PowerTerm Pro. (See *www.powerterm.com.*)

You can enter the insert or replace mode using several different commands, depending on where you want to begin entering text. Table 6-3 shows the most commonly used commands of this type. When you enter any of these commands (in command mode) you see the `--INSERT--` or `--REPLACE--` indicator at the bottom of the `vi` screen.

Table 6-3 `vi` Commands to Enter Insert or Replace Mode

Keystroke	Description
i	Begin inserting text to the left of the current cursor position.
a	Begin inserting text to the right of the current cursor position.
I	Begin inserting text at the beginning of the current line.
A	Begin inserting text at the end of the current line.
o	Insert a blank line after the line that the cursor is on, place the cursor on the new line, and begin inserting text.
O	Insert a blank line above the line that the cursor is on, place the cursor on the new line, and begin inserting text.
r	Replace one character with the next character entered.
R	Enter replace mode; all text entered will overwrite existing text beginning at the current cursor position.

Table 6-4 shows a few common editing commands that you can use in `vi`'s command mode. From the commands given here, you can deduce other similar commands. For example, if the command `10,y,y` copies 10 lines into the clipboard, the command `20,y,y` will copy 20 lines into the clipboard.

Table 6-4 Standard `vi` Editing Commands

Keystroke	Description
x	Delete one character to the right of the cursor.
5,x	Delete five characters to the right of the cursor.
d,w	Delete one word to the right of the cursor.
5,d,w	Delete five words to the right of the cursor.
d,d	Delete the current line.
D	Delete from the cursor position to the end of the current line.
u	Undo the previous command (use repeatedly to undo several commands).
y,y	Copy the current line into a buffer. (A `vi` buffer is like the Windows clipboard, but `vi` has many different buffers; this command uses a standard buffer.)
p	Paste the line(s) from the standard buffer below the current line.
J	Join the next line to the end of the current line (remove the end-of-line character at the end of the current line).

All of the commands shown so far affect the document you are editing but do not display anything as you enter the command characters. Many `vi` commands do display the text that you enter, making it easier to enter these commands. Table 6-5 shows a few of these commands, most of which begin with a colon or a forward slash. After you enter the colon or forward slash, you see the remaining characters in the command at the bottom of the screen. For each of these commands, you must press Enter to indicate that you have finished entering the command.

Table 6-5 Additional `vi` Commands

Command	Description	Example
:, w, Enter	Save the current document.	`:w`
:, w, *filename*, Enter	Save the current document as *filename*.	`:w report`
:, q, Enter	Exit `vi`.	`:q`
:, q, !, Enter	Exit `vi`, discarding any changes to the current document.	`:q!`
:, w, q, Enter	Save the current document and exit `vi`.	`:wq`
Z, Z	Save the current document and exit `vi`.	
/, *searchtext*, Enter	Search for *searchtext*.	`/annual`
/, Enter	Search again for the most recent *searchtext*.	`/`
:, !, *commandname*, Enter	Execute *commandname* and return to `vi`.	`:!dir`

Although the commands in the preceding tables may seem too numerous to memorize, you will quickly become familiar with at least the basic commands required to add or delete text and then save your changes and exit from `vi`. Other powerful `vi` features, such as complex search-and-replace tasks and integration with other Linux commands, are not discussed in detail here.

TEXT PROCESSING

The text editors presented in the previous section are used only to create or edit basic text files. These files do not contain any type of formatting, such as you would see in a word processor. To create documents that include formatting, such as bold text and multiple fonts, you must use additional utilities.

The methods that you can use to create formatted documents are of two types:

- **Graphical**, or **WYSIWYG** (pronounced "whiz-ee-wig") programs show documents on the computer screen much as they will look when printed on paper or in a Web browser (depending on the type of document you are creating).
- **Mark-up languages** define a series of codes to indicate how you want a document formatted. You can create a document using a mark-up language in any text editor, but you see the results (the effect of the codes you entered) only when you view the document in another program or print it on paper.

Mark-up Languages

The mark-up method of creating documents is older than the graphical programs that are now available. Mark-up languages are still popular with many Linux and UNIX enthusiasts. Although graphical programs may be easier to learn, they require many more system resources and often lack the flexibility of mark-up systems. If part of a document created with a mark-up language is incorrect, it is usually easy to repair or add the mark-up code that makes the document correct; in graphical systems, the user must locate a menu option that performs the needed alteration of the underlying document structure.

The best-known mark-up language is HTML, the hypertext mark-up language. HTML is used on the Web as the format for all the documents downloaded for viewing on a browser. You can choose from many different graphical programs for creating HTML documents, but

you can also use a text editor like `vi` to create an HTML document. Figure 6-8 shows an HTML document in a text editor. The figure shows the format of the mark-up codes, with each enclosed in angle brackets—for example, `<TITLE>` and `<P>`.

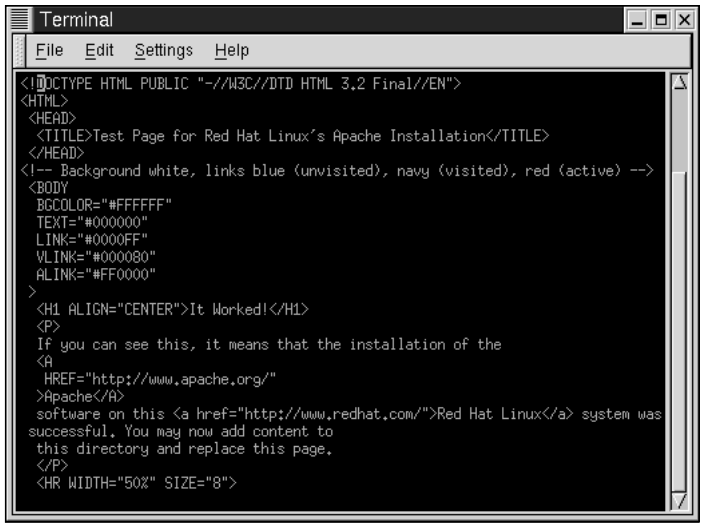

Figure 6-8 An HTML document in a text editor

The most widely used mark-up language in the UNIX and Linux world is called TeX (pronounced "tek"). **TeX** is a document-processing system that writers use to create documents—or even books—on UNIX or Linux systems. TeX is complex and requires training to use effectively, but it has a long list of features that allow it to be used effectively for projects as complex as creating scientific textbooks and software manuals.

TeX includes the capability to create macros. A **macro** is a set of commands that can be executed at one time by referring to the name of the macro. To make their work with TeX easier, writers often prefer to use a version (or package) that includes many macros. The most popular of these versions are LaTeX and TeTeX. Because of the popularity of TeX, you will often find it included on a Linux system. If you don't intend to do text processing, you can remove the TeX-related packages.

Figure 6-9 shows a **LaTeX** document in a text editor. You can see that the mark-up codes in a LaTex document begin with a backslash. As with HTML documents, you can create a LaTeX document in any text editor. You can print the resulting document on paper to see the results of the mark-up codes. You can also view the document in a graphical program called **`xdvi`**. The **`xdvi`** program displays a LaTeX-coded document as it will appear on paper. Figure 6-10 shows the **`xdvi`** program viewing a document that was originally created in a text editor using LaTeX mark-up codes.

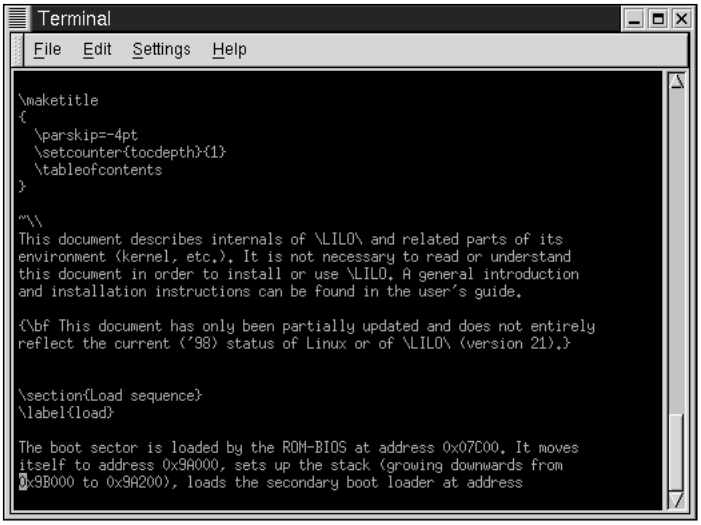

Figure 6-9 A LaTeX document in a text editor

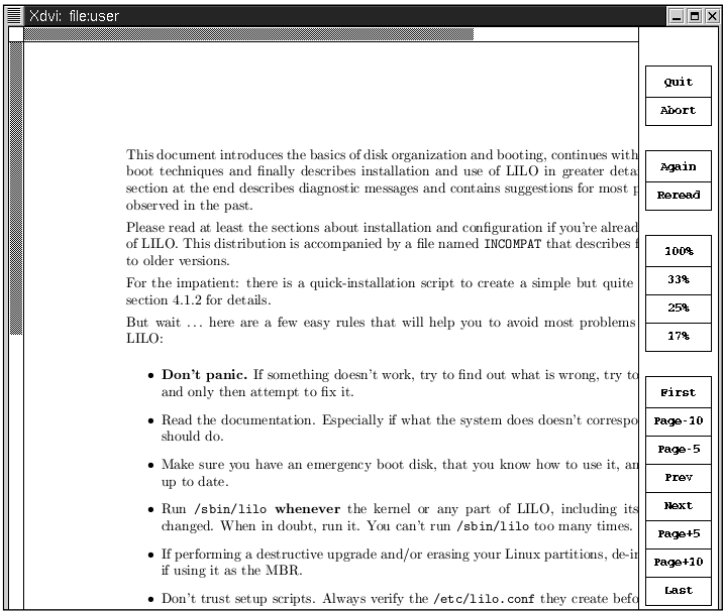

Figure 6-10 The `xdvi` program viewing a LaTeX document

Although LaTeX is a popular format for creating books and reports, it is not used much for formatting text to be displayed on a computer screen. Another mark-up language called roff is commonly used for online documents such as the online manual pages. You can use the **`troff`** and **`groff`** programs to format and display documents that are created with roff mark-up codes. (Roff rhymes with "cough." `Troff` is pronounced "t-roff" and `groff` is

pronounced "g-roff"—again, with "roff" rhyming with "cough.") Figure 6-11 shows an online Linux manual page for the `ls` command as it appears in a text editor. Notice that the roff mark-up codes begin with a period; they are different from the HTML or LaTeX codes. The `man` command converts the roff codes to formatting, such as indented lines and bold text, when a manual page is displayed on screen.

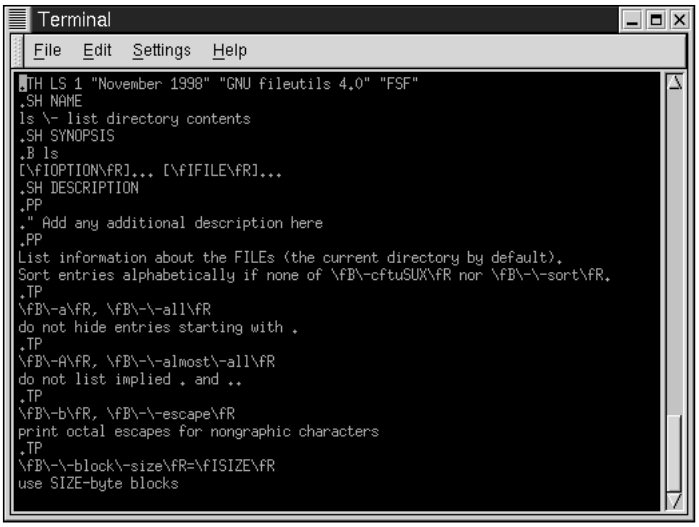

Figure 6-11 The `ls` man page file in a text editor

Some systems allow you the best of both worlds by combining a graphical system with a mark-up language. The WordPerfect for Linux product uses a WYSIWYG system for creating documents, so you don't need to learn any mark-up codes to create complex documents. But you can view the codes used internally by choosing Reveal Codes on the View menu. Figure 6-12 shows a document in WordPerfect for Linux with the Reveal Codes option selected so that the internal mark-up codes are visible at the bottom of the screen.

Controlling Fonts

A large part of formatting documents in Linux relates to fonts. This section describes several methods of controlling the fonts used on a Linux system.

The LILO program that launches the Linux kernel can control which video mode is used for the text-based display. Only a few options are available, but this flexibility is useful when you want to control the size or style of the display. The image section of the `lilo.conf` file that you learned about in Chapter 4 can contain a `vga` parameter that defines which video mode is used when the system starts. Table 6-6 shows the possible video modes and the corresponding value of the `vga` parameter.

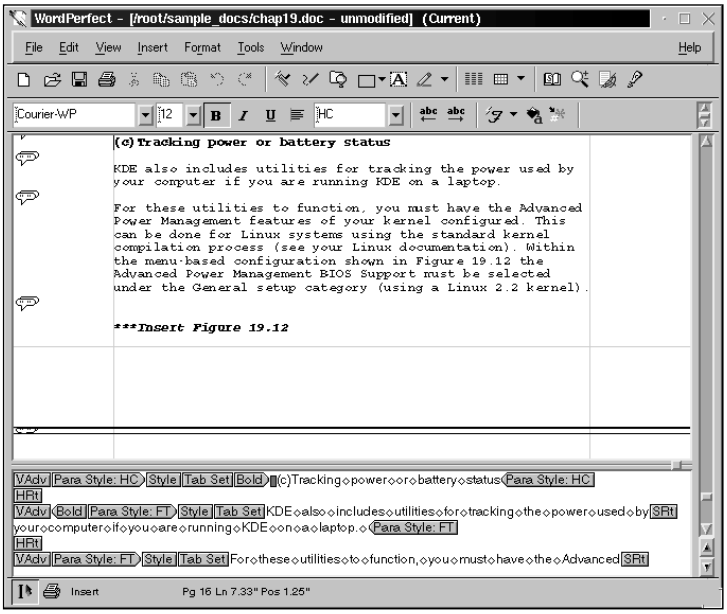

Figure 6-12 A WordPerfect for Linux document with mark-up codes visible

Table 6-6 Screen Resolutions for the `vga` Parameter

`vga` parameter value	Resolution (lines × characters per line)
0	80×25
1	80×50
2	80×43
3	80×28
4	80×30
5	80×34
6	80×43
7	40×25
8	40×28
9	132×43
a	132×43

Not all screen resolutions will be supported on all video cards. Before setting a mode, you may want to use the parameter `vga=ask` so that you can enter different values from Table 6-6 to test each until you find an appropriate choice for your system and preferences. The image section of the `lilo.conf` file with a video mode of 2 is shown in the following list:

```
image=/boot/vmlinuz-2.2.5-15
      label=linux
      root=/dev/hda3
      read-only
      vga=2
```

More important than the `vga` option for setting a character-mode font size is the ability to work with many fonts in the X Window System. Each font used in the X Window System is stored in one of the font directories named in the `XF86Config` file (described in Chapter 5). The common location for the font files is `/usr/X11R6/lib/X11/fonts`. This directory contains subdirectories for different types of fonts, such as Type1 for PostScript fonts. You can review the fonts available on the system by using the `xfontsel` program. **`xfontsel`** is a graphical program that lets you choose each aspect of a font definition (such as the font family and typeface). It then displays the corresponding font for your review. Figure 6-13 shows the `xfontsel` program with a font selected and displayed in the bottom portion of the program window.

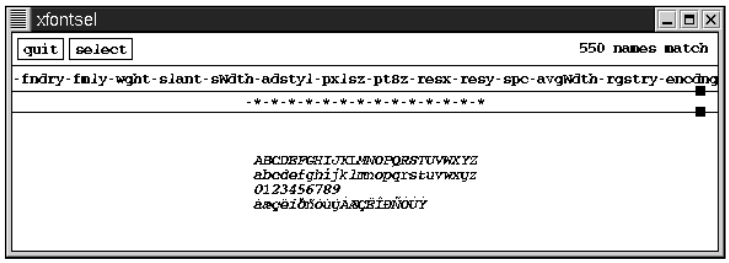

Figure 6-13 The `xfontsel` program

You can use the `xfontsel` program to select a font for a graphical program. The font that you choose can be added to a resource database file such as `.Xdefaults` (see Chapter 5), or it can be selected within the graphical program (if that option is provided by the program). Not all X fonts are appropriate for all displays. Some are very small and cannot be enlarged; others are poorer quality, suitable only for older computer systems with limited video display capabilities.

When you use Gnome, KDE, or other graphical desktop you can select the font for many applications; you can also select a default font for the desktop itself. The Gnome Control Center and the KDE Control Center both provide access to a standard font selection dialog box in which you can choose the font you want to use. (In the Gnome Control Center, use the Themes item; in the KDE Control Center, use the Fonts item.) Many graphical applications allow you to select fonts using the same standard dialog box. The font selection dialog box for the Gnome Control Center is shown in Figure 6-14.

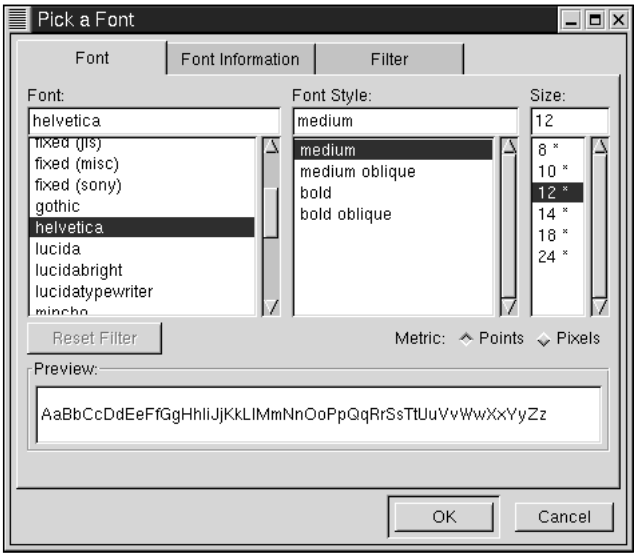

Figure 6-14 The Gnome font selection dialog box

KDE provides a Font Manager application in which you can set up the X Window System fonts that will be available in the KDE font selection dialog box. Because some X fonts may not be appropriate for your system, limiting the display of these fonts using the font manager makes it easier for you to select a font in the font selection dialog box.

Adding new fonts to a Linux system requires several steps, as outlined here:

1. Obtain the file for a new font from an Internet site of a font vendor such as Adobe. The preferred file format is `pcf`, but other formats such as `snf`, `pfa`, `pfb`, `spd`, and `bdf` are also supported by XFree86. Check the file extension of the font file to be certain that the format of the font is supported.
2. Go to the font directory specified in the `XF86Config` file, which is normally `/usr/X11R6/lib/X11/fonts`. Copy the font file into the correct subdirectory based on its type (such as `Type1`, `Speedo`, or `100dpi`). Ask the font vendor for this information if it is not apparent from the font's name.
3. Within the subdirectory where you copied the font file, locate the `fonts.scale` file. Load it into a text editor and add the font file that you copied into this directory, following the format of the other lines in the `fonts.scale` file. Increase the number on the first line of the file by one (this is the number of fonts in the file). If the font you have installed is not scalable, you can skip this step.
4. Activate the new font immediately by using the following command. (The font will also be activated automatically if you exit and restart the X Window System.)

   ```
   xset fp rehash
   ```

Altering Text Files

The mark-up languages presented previously let you create complex, professional-looking documents using a Linux text editor. Many times you'll want to modify part of a plain text file by adding, removing, or altering data in the text file based on complex rules or patterns. This kind of modification, known as **filtering**, is not possible in even a powerful text editor such as `vi` or `emacs`. To filter text files, you need to use some special Linux commands.

Linux provides many commands for filtering text files. A simple example is the `sort` command. You can use the **`sort`** command to sort all of the lines in a text file, writing them out in alphabetical order or according to an option you provide to the command. A simple example is the following, which prints a list of all users on a Linux system, sorted by the username.

```
sort /etc/passwd
```

Other options for the `sort` command allow you to merge multiple files, sorting the contents of all files; to sort based on different fields within each line of a file; and to check whether a file is already sorted.

More complex commands for altering text include a complete programming syntax to let you define how to filter a text file. The simplest of these complex commands is the `sed` command. The **`sed`** command (for *stream editor*) processes each line in a text file according to a series of commands provided by the user. The following command prints to the screen all lines of the `/tmp/names` file that contain the text `wells`.

```
sed -n '/wells/p' /tmp/names
```

The pattern between the two forward slashes (`wells` in the above example) can be very complex. In Chapter 7 you will learn about regular expressions, which you can use in a `sed` command to match complex patterns. As another `sed` example, the following command prints to the screen all lines of the `/tmp/names` file except those containing `wells`. (The `d` after `wells` indicates "delete matching lines from the output.")

```
sed '/wells/d' /tmp/names
```

A final example shows how to replace all occurrences of the pattern `wells` in the file `/tmp/names` with the string `welles`:

```
sed 's/wells/welles/' /tmp/names
```

The syntax of the `sed` commands can become very complex. Other programs such as `awk` and `perl` are also often used to filter text. Both **`awk`** and **`perl`** are full programming languages that developers use to create scripts for working on text files. (Both `awk` and `perl` are also used for many other types of tasks besides filtering text files.)

CHAPTER SUMMARY

- The Linux shell operates like any other programs on a Linux system. Many types of shells are available, such as `bash`, the Korn shell, and the C shell. All shells are used primarily for launching other programs, including system administration utilities. All shells also provide ease-of-use features such as a history list and tab completion. Many different scripts are used to initialize a shell when it is launched.
- You can customize a shell by using aliases to assign new strings to information that you enter at the shell prompt. Environment variables provide values that any program can access. You can view the value of an environment variable or set up a new environment variable from the shell prompt.
- Linux systems include numerous text editors. The most widely available is called `vi`. The `vi` editor is powerful, but requires that you memorize a series of commands to use it. Graphical editors are included on modern Linux desktops, but knowledge of `vi` remains a critical system administrator skill.
- Text files can be created using WYSIWYG word processors or mark-up languages such as LaTeX and roff. Files that use a mark-up language can be created in any text editor. To see the effect of the mark-up codes, you must use another program or print the file to paper. Several programs are available to filter lines in text files based on simple or complex rules. The `sed` command is one example of a text-filtering program.

KEY TERMS

`.bashrc` — A configuration script that is executed each time the user starts a `bash` shell.

`.profile` — A configuration script that can be located in each user's home directory. A script that is executed each time any user on the system starts a `bash` shell. This script is not included by default on all Linux distributions, but can be created if needed.

`/etc/profile` — A script containing configuration information that applies to every user on the Linux system.

alias — A string of characters that the shell substitutes for another string of characters when a command is entered.

awk — A programming language that developers use to create scripts for working on text files and completing other complex tasks.

bang — In Linux jargon, an exclamation point character

`bash` — Short for *Bourne Again shell*, an enhanced and extended version of the Bourne shell created by the GNU project for use on many UNIX-like operating systems. `bash` is the default Linux shell.

Bourne shell — The original shell for UNIX, written by Stephen Bourne.

C shell — A shell developed by Bill Joy in the 1970s. He focused on adding easy-to-use features for interactive work at the shell prompt. (Most of these features were later added to the `bash` shell as well.) The C shell is not popular for shell programming because its syntax is more complex than that of the Bourne, `bash`, and Korn shells.

command interpreter — A program that accepts input from the keyboard and uses that input to launch commands or otherwise control the computer system.

`echo` — Command used to print text to the screen.

environment variables — Settings, or values, available to any program launched by a particular user. Each user has a separate set of environment variables available to programs launched by that user.

`export` — Command used to make a newly created environment variable available to other programs running in the same environment.

filtering — The process of adding, removing, or altering data in the text file based on complex rules or patterns.

`groff` — A command used to format and display documents that are created using roff mark-up codes.

`history` — A command used to display all of the stored commands in the history list.

history feature — A feature of the shell that records in a list (the history list) each of the commands that you enter at the shell prompt.

history list — A list that contains the most recently executed commands. (Normally at least 100 commands are included in the history list.)

Korn shell — A revision of the Bourne shell that includes the interactive features of the C shell but that maintains the Bourne shell programming style. The Korn shell was written by David Korn.

LaTeX — A version of the mark-up language TeX that includes numerous macros for easy document creation.

`ln` — Command used to create a symbolic link.

macro — A set of commands that can be executed as one by referring to the name of the macro.

mark-up languages — Computer languages that define a series of codes indicating how to format a document.

modal editor — A text editor that uses multiple modes for editing text and entering commands to apply to that text.

`PATH` — An environment variable containing a list of directories on the Linux system that the shell searches each time a command is executed.

perl — A programming language that developers use to create scripts for working on text files and completing other complex tasks.

`sed` — A command used to process each line in a text file according to a series of commands provided by the user.

`set` — Command used to display a list of all environment variables defined in the current environment.

shell — The command interpreter in Linux.

shell prompt — A set of words or characters indicating that the shell is ready to accept commands at the keyboard.

`sort` — A command used to sort all of the lines in a text file, writing them out in alphabetical order or according to options provided to the command.

symbolic link — A file that refers to another filename rather than to data in a file.

tab completion — A feature of the shell that lets you enter part of a file or directory name and have the shell fill in the remainder of the name.

TENEX/TOPS C shell (**TC shell**) — An enhancement of the C shell. This is the version of the C shell that is commonly used on Linux systems.

TeX — A document processing system that writers use to create large and complex documents on UNIX or Linux systems.

`troff` — A command used to format and display documents that are created using roff mark-up codes.

WYSIWYG — A characteristic of programs that show documents on the computer screen much as they will look when printed on paper or in a Web browser (pronounced "whiz-ee-wig").

`xdvi` — Program used to display a LaTeX-coded document as it will appear on paper.

`xfontsel` — Program that lets the user choose each aspect of a font definition (such as the font family and typeface) and then displays the corresponding font for review.

REVIEW QUESTIONS

1. The default shell used by Linux cannot be altered. True or False?
2. When logged in as `root`, the shell prompt normally changes to display:
   a. A `%` character
   b. A `#` character
   c. The root directory
   d. A `$` character
3. The main function of a shell is to:
   a. Track kernel resources for `root`
   b. Provide a convenient programming environment
   c. Complement desktop interfaces
   d. Launch programs entered at the shell prompt
4. Name four different shells and briefly describe the differences between them.
5. Tab completion is useful when you need to:
   a. Repeat a previously used command
   b. Reinitialize the X Window System font list
   c. Enter long filenames or directory names at the shell prompt
   d. Create a brief shell program
6. The `history` command is used to
   a. Display a list of previously entered commands
   b. Execute a previously used command
   c. Change the environment variable controlling tab completion
   d. Edit an existing text file

7. Entering the command `!fr` would do the following in the `bash` shell:
   a. Cause an error because the command name is incomplete
   b. Execute the most recently executed command that began with `fr`
   c. Execute the `free` command to display system memory
   d. Search for the pattern `fr` in the `vi` editor
8. To have a command executed each time any user logged in to the Linux system, you would place the command in which one of these files:
   a. `/etc/profile`
   b. `/etc/.profile`
   c. `~/.profile`
   d. `/etc/bashrc`
9. If a `.bashrc` file is found in a user's home directory, the systemwide `/etc/bashrc` script is not executed. True or False?
10. If a directory contains the filenames `micron` and `microscope`, and you enter `micro` and press Tab, what happens?
    a. The shell prints all matching names, `micron` and `microscope`.
    b. The shell fills in the first alphabetical match, `micron`.
    c. The shell beeps.
    d. The `micron` command is executed.
11. Describe the difference between an alias and a symbolic link.
12. Which of the following is a correctly formed alias for executing the `mv` command?
    a. `alias ren mv`
    b. `alias ren=mv -i`
    c. `alias mv=ren`
    d. `alias ren="mv -i"`
13. Which command is used to create a symbolic link?
    a. `sh`
    b. `ln`
    c. `set`
    d. `sed`
14. The command `echo $HOME` will display:
    a. The word `HOME`
    b. The current user's default shell
    c. The value of the `HOME` environment variable
    d. A prompt requesting a home directory path

15. Describe the purpose of the `PATH` environment variable.
16. The `export` command is used to make an environment variable available to other programs. True or False?
17. Name at least three nongraphical text editors that may be included with a Linux distribution.
18. Knowledge of the `vi` editor is considered an essential skill because:
    a. Memorized `vi` commands correspond to other Linux command options.
    b. The `vi` editor is virtually always available to complete system administration tasks.
    c. Other editors are not as reliable or easy to use.
    d. The developer of `vi` also developed part of Linux.
19. Describe the result within `vi` of pressing the following keys:

    ```
    itest<Esc>yyp:wq<Enter>
    ```
20. To view the formatted appearance of a LaTeX document, you would use the following command:
    a. `xdvi`
    b. `groff`
    c. `TeTeX`
    d. WordPerfect for Linux
21. Describe the difference between a WYSIWYG program and a document containing mark-up codes.
22. The `lilo.conf` file can contain the `vga` option, which controls:
    a. Whether the X Window System is available after the system boots
    b. Which display mode is used on the character-mode console
    c. The action of the virtual `grep` archive
    d. How many colors can be displayed on the system
23. Fonts for the X Window System are normally stored in which directory?
    a. `/usr/X11/xdm/fonts`
    b. `/usr/X11R6/fonts`
    c. `/etc/X11/xinit/fonts`
    d. `/usr/X11R6/lib/X11/fonts`
24. Filtering text files refers to removing lines matching a certain pattern. True or False?
25. Name three programs that can be used to filter text files in Linux.

HANDS-ON PROJECTS

Project 6-1

In this activity you use tab completion to explore the Linux file system and alter an environment variable within the shell. To complete this activity you should have a working Linux installation with a valid user account. The filenames described in this activity are taken from a Red Hat Linux installation, but the steps will work on other Linux versions as well.

1. Log in to Linux using your username and password.
2. If you are using a graphical environment, open a terminal window so you have a shell prompt.
3. Change to the directory `/bin` using the command **`cd /bin`**.
4. List the shells that are installed on the system using the command **`ls *sh`**. Can you recognize all of the shells listed?
5. Change to the directory `/etc` using the command **`cd /etc`**.
6. Type the command **`ls -l host`** but don't press Enter.
7. Press the **Tab** key twice. The first time you press Tab the shell beeps. The second time it displays a list of files in `/etc` that begin with `host`.
8. Type **`s.`** (so that the command line contains `ls -l hosts.`), but don't press Enter.
9. Press the **Tab** key twice. The shell beeps and then displays all the files in `/etc` that begin with `hosts.` (including the period). The list is shorter than the output of Step 7 because you added more characters to search for.
10. Type the **`a`** and press **Tab**. The shell fills in the full filename so that the line reads `ls -l hosts.allow`.
11. Press **Enter** to complete the `ls` command that the Tab key finished filling in.
12. Change to your home directory by entering the command **`cd`**.
13. Enter the command **`!ls`** to execute the most recently used `ls` command, which you entered in Step 11. Why does the command display an error now?
14. Enter the command **`echo $PS1`** to display the format of the standard shell prompt.
15. Enter the command **`man bash`** to view the manual page for the `bash` shell.
16. Enter the text **`/\\W`** to search for the string `\W`, which is part of the `PS1` definition you saw in Step 14.
17. Use the arrow keys to review the list of parameters that you can use to redefine the `PS1` environment variable. Locate the `\d` option.
18. Press **`q`** to exit the man page viewer.
19. Enter the command **`export PS1="\d$PS1"`**. What happened? What does the `$PS1` at the end of the command indicate?
20. Enter the command **`bash`** to start a new shell. How does the shell prompt change? Why?
21. Enter the **`exit`** command to leave the new shell you started in Step 20. How does the shell prompt change? Can you explain this?

Project 6-2

In this activity you work with the `vi` editor to make a change to a shell start-up script. To complete this activity you should have a working Linux installation with a valid user account.

1. Log in to Linux using your username and password.
2. If you are using a graphical environment, open a terminal window so you have a shell prompt.
3. Enter the **`pwd`** command to make certain you are in your home directory.
4. Enter **`vi .bashrc`** to display the `.bashrc` file in the text editor window.
5. Press **Shift+G** on the keyboard to move to the end of the file.
6. Press the **o** key to start inserting a new line of text.
7. Type the text **`TEST_VAR="This is a test"`** and press **Enter**.
8. On the next line type the text **`export TEST_VAR`** and press **Enter**.
9. On the next line type the text **`alias tv="echo $TEST_VAR"`**.
10. Press **Esc** to return `vi` to command mode.
11. Hold down the **Shift** key while you press **Z** two times to save the file and exit `vi`. (You can also use the other methods shown in the chapter if you prefer.)
12. Type **`tv`** and press **Enter**. What is the result?
13. Start a new shell by entering the command **`bash`**.
14. Type **`tv`** and press **Enter**. What is the result? Why?
15. Enter the **`exit`** command to exit the additional copy of `bash` that you started in Step 13.
16. Enter the command **`vi .bashrc`** to begin editing the same file as in previous steps.
17. Press the **j** key repeatedly until the cursor is located on the line containing `TEST_VAR="This is a test"`.
18. Type **3**, then press **d** twice to delete the three lines that you entered.
19. Type a colon (**:**), type **w**, type **q**, and then press **Enter** to save the file and exit.

Project 6-3

In this activity you explore various font issues on the Linux system. To complete this activity you should have a working Linux installation with a valid user account and the X Window System running (using any window manager or desktop interface). The commands described in this activity are included in Red Hat Linux; other versions of Linux may not have all of the same utilities installed by default.

1. Log in to Linux using your username and password.
2. If you are using a graphical environment, open a terminal window so you have a shell prompt.

3. Enter the command **sed -n '/vga/p' /etc/lilo.conf** to see if the LILO configuration on your system includes any preset video modes for the console character display.
4. Enter the command **vi /etc/lilo.conf** to review the LILO configuration file. The file appears on screen.
5. If anything was output by the command given in Step 3, can you locate that line in the `lilo.conf` file within `vi`? If nothing was output, can you identify where the `vga` parameter would go (within an `image` section)?
6. Enter the command **:q!** to exit `vi` without saving any changes that you inadvertently made.
7. Change to the X `fonts` directory using the command **cd /usr/X11R6/lib/X11/fonts**.
8. Use the **ls** command to list the subdirectories within the `fonts` directory.
9. Change to the `75dpi` subdirectory using the command **cd 75dpi**.
10. Review the list of fonts included in this directory using the command **more fonts.dir**. Press the **Spacebar** to advance the list of fonts; press **q** to exit the listing.
11. Start the `xfontsel` program by entering **xfontsel**.
12. Explore the buttons and drop-down lists provided in `xfontsel`. How do they correspond with the information in the font listing you saw in Step 10?

CASE PROJECTS

1. You are the system administrator for a large travel agency where you manage Linux workstations for about 70 employees. The employees use the workstations to access several types of text-mode reservations systems. They also use a browser on the Linux systems to review Web sites related to travel and travel destinations, and to exchange e-mail with clients. Jill, one of the more technically inclined employees, approaches you. She has some requests and recommendations.

   Some of the programs used by the employees require that certain environment variables be set. Jill suggests that the environment variables be set up in some type of automatic way so that users don't have to enter the values each time they start the program in question. Jill heard about aliases from a friend and asks if that might be a good option. Is it? What other options would you consider? Would you consider placing something in each user's home directory, or would a file that applied to all users on the system be preferable?

2. Jill has used `vi` for several different tasks, but she doesn't use it often. She asks if you could install another text editor of some type so that she can edit files without using `vi`. How do you respond? Do you see any reason not to grant her request? Given that all of the users are working in a graphical environment (with a Web browser as well as text-based applications), what text editors might you consider recommending to Jill? Should other users be informed about text editor options if they don't already know about them?
3. Jill confides that one of the older travel agents has trouble reading the small characters on the terminal windows that are displayed in the graphical environment. You know that these can be changed. Research how to make these changes in a terminal window for the desktop interface on the Linux systems you work with, and then determine how to make the change permanent (so that large characters are always used). (The steps required depend on which Linux distribution or desktop interface you are using.)